63-2-4

MEMORANDUM
RM-3447-PR
JANUARY 1963

PROGRAMMING LANGUAGES AND STANDARDIZATION IN COMMAND AND CONTROL

J. P. Haverty and R. L. Patrick

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-3447-PR
JANUARY 1963

PROGRAMMING LANGUAGES AND STANDARDIZATION IN COMMAND AND CONTROL

J. P. Haverty and R. L. Patrick

This research is sponsored by the United States Air Force under Project RAND — Contract No. AF 49(638)-700 — monitored by the Directorate of Development Planning, Deputy Chief of Staff, Research and Technology, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force. Permission to quote from or reproduce portions of this Memorandum must be obtained from The RAND Corporation.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA

PREFACE

This Memorandum is the final report on a study of programming languages undertaken by The RAND Corporation. The rapid growth in the design, publication, and use of programming languages in military computer applications has raised several issues which affect current and projected Air Force computer-based systems.

The purpose of the study was to identify and discuss the two major issues raised by the use of programming languages: 1) what benefits programming languages offer, and 2) what gains are to be realized by standardizing on a programming language.

This Memorandum is addressed to the user of electronic data processing systems and, in particular, to the Air Force groups who provide guidance and planning in electronic data processing, such as the Air Force Directorate of Data Automation and the Electronic Systems Division.

SUMMARY

This Memorandum presents the findings of a study of common programming languages with emphasis on Air Force applications, particularly in the area of Command and Control. The objectives of the study were to assess the state-of-the-art in programming languages and to discuss the consequences of standardizing on a programming language.

First, the Memorandum emphasizes that the almost complete lack of facts and the absence of a standard glossary of terms are major obstacles to any study and evaluation of programming languages. Given these limitations, the Memorandum examines the pros and cons of current programming languages, management and design considerations, the urgent need for measures of performance, and some desired developments in compilers.

Next, the question of standardizing on a programming language is discussed. The current programming language area is evaluated in terms of three factors--a sufficient base of existing lower-level standards, mature technical development, and extensive user experience. The evaluation indicates that it is not now appropriate to establish a standard programming language; however, planning for standardization should be started immediately. Measures short of complete standardization--e.g., partial standardization, standardization on a communication language--may achieve most of the benefits of full standardization and should be examined.

The major conclusions of the study are:

1) Much work remains to be done in the computer field in establishing basic standards (below the programming language level) before we can achieve the benefits of a standard programming language.

Particularly needed are: a data processing glossary, magnetic tape recording formats, and character sets.

2) Partial standardization measures should be considered and evaluated now.

3) A central Air Force agency is needed to establish measures of performance, to plan for standardization, and to plan and coordinate computer selection.

4) Programming languages, while offering some important advantages, are still in a relatively immature state of technical development. The Air Force should take the lead in directing and supporting research efforts to improve the state-of-the-art.

5) An indoctrination program in computers and computer programming for Air Force officers is required now, and should be supplemented as soon as is feasible by courses in the established USAF schools and colleges. In addition, a training program is urgently needed to develop a cadre of skilled programmers in the Air Force. A staff study should be instituted to formulate a detailed course of action for establishing these programs.

CONTENTS

PREFACE iii

SUMMARY v

Section

I. INTRODUCTION 1
   Background of the Study 1
   The Problem of Definition 2
   The Programming Problem 7
     Definitions 7
     Programming Languages and the Development of Large Computer-Based Systems 9

II. PROGRAMMING LANGUAGES 12
   Brief History 12
   Pros and Cons of Current POLs 14
     Training 14
     Reprogramming and Computer Conversion 15
     Program Checkout (Debugging) 17
     Programming Costs 18
     Better Understanding by Higher Management .   19
   Programming Language Selection 19
     Management Considerations 20
     Design Considerations 21
   Measures of Performance 29
     Deficiencies in the Current Program 29
     Measurement Program Outline 30
     Performance Parameters 31
     Three Measurement Studies 33
   Some Desired Compiler Developments 33
   Summary 35

III. THE STANDARDIZATION QUESTION 37
   Background 37
   Definitions 38
   Requirements for a Standard POL 40
     A Sufficient Base 40
     Mature Technical Development 40
     Extensive User Experience 42
   Planning for Standardization 42
   Partial Standardization 43
   Summary 43

IV. CONCLUSIONS 45

Appendix

THREE MEASUREMENT STUDIES 53
   B. F. Goodrich Company Study 53

Study of Programming Systems for the IBM 650 55
BUSHIPS COBOL Evaluation 58

BIBLIOGRAPHY 63

I. INTRODUCTION

BACKGROUND OF THE STUDY

In January 1962, the Computer Sciences Department of The RAND Corporation decided to undertake a study of programming languages. The debate on the capabilities of programming languages and how they are related to the issue of standardization had reached proportions which indicated that a formal study should be initiated. Furthermore, discussions engendered by the Institute for Defense Analyses report, TM 61-12, "Computers in Command and Control," indicated that the time for such a study was at hand.

One of the first objectives of the study was to gather known facts and to evaluate previous work in an attempt to clarify the state-of-the-art in programming languages and to classify languages with regard to their technical characteristics, advantages, and disadvantages. Finally, the study was to address itself to the question of programming language standardization; in particular, what is to be gained or lost if we standardize--either now or later.

Such an undertaking is especially difficult in the computing field since the number of facts available on programming languages is quite small. An extensive literature search* produced volumes of material on feasibility studies, proposed rather than accomplished work, detailed narrow implementations and, too often, opinions stated as facts. The search failed to produce any significant quantity of controlled experiments or operational measurements. Particularly lacking was any material concerning the relative advantage of accomplishing a task through the use of alternative means. Thus, it should be stressed, we had to use more than the usual amount of technical opinion in this study and therefore some of the conclusions and assertions rely primarily on judgments made in the absence of substantive data.

*See Bibliography

THE PROBLEM OF DEFINITION

One of the most glaring deficiencies in the computing field came to light early in the study--the lack of a standard glossary of terms. In reviewing the literature we were impressed with the confusion that can result because of the variety of meanings that certain technical terms apparently have.

To try and reach a common level of understanding with our readers, we have listed pertinent definitions as appropriate for this section and succeeding ones.

LANGUAGE	A set of symbols, together with rules for their grouping into meaningful combinations, used by one participant to describe a process so that another participant may understand the process. Natural languages have several additional groupings: symbols into words, words into sentences, and sentences into paragraphs. The rules for a natural language cover syntax, grammar, and composition.
MACHINE LANGUAGE	The wired instructions directly interpreted by the computer during execution time. If the adjective "symbolic" is used to modify the phrase "machine language," reference is made to a "humanized" version of machine language. This humanization consists primarily of substituting a mnemonic combination of standard characters for the machine language codes and storage addresses interpreted by the hardware. In a "symbolic assembly" this substitution is generally on a one-for-one basis; i.e., one handwritten symbolic instruction results in one machine instruction.

PROGRAM*	Any process, which, when reduced to machine language, completely describes a procedure in a form such that a computer can interpret the procedure and perform the desired processing.**
ASSEMBLY PROGRAM	A computer program which receives input in the form of symbolic language, translates this in accordance with predetermined rules, and produces output in the form of machine language. Assembly programs permit a programmer to assemble with his program other symbolic programs (usually called library routines). Thus, frequently-used programs (decimal to binary conversion, sin x, log x, etc.) need be programmed only once and then can be placed in a library available to other programmers. The use of an assembly program requires detailed knowledge of the inner workings of the computer. In addition, the programmer must be attuned to the capacity of the computer being used--both for the assembly process and the ultimate application. The programmer must know the machine configuration in detail so that the physical limits of the device are not exceeded.
SOURCE LANGUAGE	The language used in the statement of a process in machine-readable form as the <u>input</u> to a computer translator process.

*Care must be taken not to confuse a computer program with a plan or schedule for the attainment of some objective.
**A frequently-used synonym is "computer routine."

OBJECT LANGUAGE	The *output* of a translation process. A computer plus a translation program receives source language as input and outputs object language. The collection of output is called, among other things, the object program. The machine on which this translation process takes place is properly described as the source machine or source computer. Similarly, the machine for which the program is being produced is called the object machine or object computer.
TRANSLATOR	The mechanism (program, human effort, electronic device, etc.) by which the input or source language is transformed into the output or object language. More restrictively, the program for converting symbolic language to machine language.
COMPILER	A program for a specific source computer which translates input in the form of source language into output in the form of object language. The object language may be symbolic language for a subsequent assembly process (sometimes referred to as an "intermediate language"). A compiler differs markedly from an assembly program in that the translation from source language to object language is usually *one* to *many*; i.e., one statement results in many machine language instructions. In addition, a compiling program handles most of the normal burden of primary storage allocation and generally handles much of the burden of secondary storage allocation.

PRIMARY STORAGE	The memory system which is most closely coupled to the arithmetic and logical element of the computer. In large machine technology this storage has the characteristics of uniform access (in time) to all cells, high cost, and high performance. It usually takes the form of magnetic cores.
SECONDARY STORAGE	A non-homogeneous grouping used to denote machine readable storage media whose characteristics are: access time (to any particular storage location) not uniform, access time significantly greater than primary storage, and considerably lower cost than primary storage. The more predominant types are: punched cards, punched tape, magnetic drums, and magnetic disks. Files stored on secondary storage can frequently be physically removed from the computer and stored semi-permanently in some other place.
COMMUNICATION LANGUAGE	A language structure complete with conventions, syntax, and character set, used primarily for conveying knowledge of processes between two participants. A translation program which processes this source language and outputs machine language does not necessarily exist.
PROCEDURE ORIENTED LANGUAGE (POL)	Denotes any source language which was derived with a particular restricted class of problems in mind.* The

*Not to be confused with Problem Oriented Language, a term sometimes used synonymously and sometimes used (confusingly) to describe yet another type of higher-level language.

resulting source language is then compiled and the final object program will, when checked out, instruct a general purpose digital computer so that it generates the solution to the problem at hand. The most widely used example of this is FORTRAN, used for scientific and engineering applications (formula evaluation).

Some other examples are COBOL, business data processing (file processing and handling); JOVIAL, Command and Control (extensive data manipulation, many individual subprograms integrated into one large program); and IPL-V, heuristic programming (list structure processing).

COMMON LANGUAGE

A programming language which is used for the solution of like problems in more than one physical installation. A common language does not necessarily apply to more than one problem area. A common language generally is used on several machine types of the same manufacturer but is not necessarily used on machines of different manufacturers even though they are in use in the same problem area. The most prevalent common language from recent history is the SAP language for the IBM 704. The most prevalent present common language is the SPS assembly language for the IBM 1401. The most prevalent common language that is used on many types of computers is FORTRAN.

	A language becomes common through convention. Sometimes a language is voluntarily adopted as a common language before it sees service, but normally, a language retains its common status only through performance. Any time the factors indicating a change outweigh the advantages of commonness, the language loses its precarious status. Common languages almost always exist in dialects. The dialects are usually a function of the configuration of the machine on which the translator is implemented, and the particular requirements of an individual user.
REAL-TIME PROCESS	A process where the computer exists on-line in an environment which is not and cannot be made subservient to computer control. Usually a continuous (on-going) physical process is involved. If the time available for solution is only slightly larger than the time required for computation, the environment is said to be real-time.

THE PROGRAMMING PROBLEM

In order to examine the programming problem and the role of programming languages, it is essential to discuss the major features of the programming process by defining its component sub-processes.

Definitions

PROBLEM FORMULATION	The process by which the problem solution is clarified, detailed, and then expressed in precise form. The

output of this process is the "design blueprint" which may take the form of program specifications, flow charts, and coding specifications. Currently, this is a human-to-human process. (POLs will not materially affect the elapsed time, cost, etc., of this process. They may help indirectly in formalizing a "design or specification language.")

CODING

The process of translating from the "design blueprint" to the source language used. In some cases the difference between the two is small (e.g., the difference between a SAGE Coding Specification and JOVIAL) and hence the elapsed time for this translation phase is small. In other cases, the difference is great (e.g., a verbal flow chart and symbolic assembly language). This is the part of the programming process that POLs were designed to speed and improve.

CHECKOUT

The process of establishing that the program (or procedure) in the computer is accurately performing the desired processing. The opportunities for differences or errors abound because of: transmission error (I meant to write "<" and wrote ">"); interpretive errors (I thought the number of aircraft would never exceed 1000); performance errors (I should have cleared this field to zero first); transcription errors (the key punch operator punched a "$" instead of "*"); etc. POLs may help here because they are closer to the design or specification language.

However, they still offer the same opportunity for these types of errors.

DOCUMENTATION	A process that should properly cut across the above three processes, describing what has transpired for the edification of the author, his supervisor, or his replacement. POLs hope to reduce the number of forms of the documentation and thereby keep it current.
SOFTWARE	One definition: "A colloquial term for any program or method of use which can perform hardware functions."* Current definition: The package of programming support or utility routines which is provided (or is available with) a given computer.** The package generally includes: an assembler, a compiler, an operating system (or monitor), debugging aids, and a library of subroutines.

Programming Languages and the Development of Large Computer-Based Systems

Much of the sound and fury over programming languages is the result of zealous proponents touting them as the solution to the "programming problem." This "problem" arose when it became apparent that programming had become the pacing factor in the development of large military data processing systems, such as SAGE. The claim for a solution is, of course, nonsense but because it is a persistent one, we must clarify and discuss "the programming problem."

The development of large computer-based Command and

*Reference Manual Glossary for Information Processing, International Business Machines, 1962.

**Apocryphally, "The package which attempts to make up for all the deficiencies and omissions in the hardware design."

Control systems* represents a significant break with the traditional development cycle of weapon systems. The most significant difference is that in weapon systems development the design problems of minimizing costs and elapsed time and of maximizing performance were centered on the prime hardware. Support requirements were (and probably rightly so) considered to be less critical than the development and procurement of the prime equipment. This approach was totally inappropriate to the development of Command and Control systems. The development of SAGE (and subsequently, other large systems) revealed the following markedly different characteristics.

a. Hardware was no longer the sole significant lead-time item.

b. Programming--especially the problem formulation and testing phases--was a long lead-time item.

c. Support requirements were recognized as significant factors in costs and performance.

d. An evolutionary, rather than concurrent, approach to development and installation was desired.

Thus, the "programming problem" was the recognition of a new and critical lead-time item. Unfortunately, there were no substantial across-the-board improvements available to reduce lead-time. However, in one phase of the programming process--the coding phase--efforts were underway to achieve greater efficiency. Coding represents roughly 30 per cent of the elapsed time in large programming tasks and hence gains here, while not exciting, would be worthwhile. Nevertheless, it must be recognized that POLs are at best only a partial attack on the total "programming problem."

*For a more complete discussion, see the article, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," W. A. Hosier, IRE Transactions on Engineering Management, June 1961.

The selection of evolutionary development was an admission of our limited understanding of the principles for system design and development in the Command and Control area. Moreover, this stepwise (evolutionary) approach forced a new requirement on the development of the computer programs; namely, adaptability to continuous change and upgrading. Currently, it is hoped that POLs will offer reduced lead-times in incorporating system improvements and additions. There is no concrete evidence of this to date, since no Command and Control system using a POL has reached this phase.

In summary, POLs offer hope of making some headway on the programming problem but in no sense mark a solution or spectacular advance.

II. PROGRAMMING LANGUAGES

BRIEF HISTORY

The history of programming languages is a short one. The first widely known efforts in this area date from about 1954--the cooperative Project for the Advancement of Coding Techniques (PACT) on the West Coast, IBM's FORTRAN, and Univac's A-2. These were attempts to attain a new level of capability over symbolic assembly techniques and coincided with the realization that the limited supply of trained programmers was unable to meet the sharply increased demand created by the new IBM 700 series and Univac 1100 series machines. Hence, a substantial effort appeared necessary to increase programmer productivity. This, incidentally, was a recognition of the fact that programmer time was becoming the scarce resource--a reversal from earlier days when machine time was the scarce commodity.*

From about 1955 until early 1958, these first programming languages were subjected to varying degrees of usage. FORTRAN particularly received extensive use and its success encouraged additional efforts across the country. In May 1958, a joint ACM-GAMM group met in Zurich to define an algebraic programming language. The formal effort to devise a "standard" algorithmic language culminated in the publication of ALGOL 58 and gave strong impetus to the development of several new programming languages. In rather quick succession came ALGOL derivatives: NELIAC, JOVIAL, MAD, and others. Most of these languages were developed by users with specific needs and requirements: NELIAC--Naval Tactical Data System, JOVIAL--Strategic Air Command System, MAD--University of Michigan Computer Center, etc. The fact that these languages were being developed for computer applications with differing constraints was masked from the

*Typically a programmer's work was rigorously desk-checked to reduce the number of machine runs.

uninitiated. To many observers it appeared that this was senseless proliferation.

About the same time, however, a further complicating factor arose--the designing of programming languages became fashionable. Now the proliferation of languages increased rapidly as almost every user who developed a minor variant on one of the early languages rushed into publication, with the resultant sharp increase in acronyms. In addition, some languages were designed practically in vacuo. They did not grow out of the needs of a particular user, but were designed as someone's "best guess" as to what the user needed (in some cases they appeared to be designed for the sake of designing).

These languages were aimed at a particular portion of the broad spectrum of computing applications--and frankly, not a very large portion (perhaps 30-40%). However, human nature--and salesmanship--decreed that many of the developers of these languages lay claim to more and more of the spectrum--in effect, saying "it can be used in other parts of the spectrum."* To the casual observer (and naive believer of claims) it appeared that there was a growing proliferation of overlapping POLs and that standardization was the only answer to avoid what was frequently called "The Babel of Languages."

In summary, the effort to develop programming languages to increase programmer productivity is barely eight years old. Several of the more prominent languages have been in full use only during the last few years. Advances have been made, but the solid achievements have been far outdistanced by the wild claims of the marketing departments and those suffering from the "publish or perish" syndrome. We pay a penalty in disappointment when the claims prove vacuous, and we experience increased difficulty in identifying a

---

*That is, I <u>can</u> haul hay in my new Ford (but I would consider this a last resort).

solid advance in capability. There have been too few individuals abstracting from experience, generalizing, and setting down the fundamentals of programming. Each design, however trivial, appears to be a new and unique stroke of original work on the part of the experimenter. We have been unable to locate a single text on the fundamentals of programming (not coding for a particular machine) or a cogent series of documentary examples depicting the solution of even the standard problems of programming. Perhaps these, too, await a glossary.

PROS AND CONS OF CURRENT POLS

The claims for POLs are many and varied, but the most important fall into five general areas: training, reprogramming, debugging, programming costs, and increased management understanding.

Training

The genesis of POLs lies in the shortage of trained people. The claim is often made that by using POLs the training requirements are eased and, since the POL is a language close to the user's language, the user "gets on the air" quicker. However, it should be recognized that the user's depth of understanding of the programming process is often exceedingly shallow. Many of the important principles of programming are initially shielded from him and must be discovered (often painfully) over time.

On the other hand, if the user is taught a symbolic assembly language, he takes longer to make his first run because he must learn many of the characteristics of the machine and machine language coding. But, with this machine language base, learning a POL is usually easier and the depth of understanding far greater. Probably the amount of training and experience required to become a proficient professional programmer is independent of whether a compiler or an assembler is being used.

We recognize that one or the other of these modes of training may be more suitable for any given user--we simply wish to point out that POLs, in and of themselves, do not reduce the training requirements for professional programmers. Lest the training in a POL should appear to be too easy, it should be noted that the Commercial Translator (COMTRAN) manual runs to over 300 pages with much of the content devoted to rules and conventions.

Reprogramming and Computer Conversion

The claim that intrigues the experienced user is perhaps best summarized by the following excerpt from the 1961 Eastern Joint Computer Conference Panel Discussion.*

> ...a computer user, who has invested a million dollars in programming, is shocked to find himself almost trapped to stay with the same computer or transistorized computer of the same logical design as his old one because his problem has been written in the language of that computer, then patched and repatched, while his personnel has changed in such a way that nobody on his staff can say precisely what the data processing job is that his machine is now doing with sufficient clarity to make it easy to rewrite the application in the language of another machine.

This is probably an overstatement, but the problems of reprogramming and computer conversion have always been a source of worry to installation managers. Any reprogramming is viewed with alarm because of the generally poor record of programmers as documenters and the thought of attempting to reprogram a routine in the absence of the original programmer is horrifying. The "poor documentation" syndrome carries over into the problem of computer conversion--a problem which some facility managers aggravate by ordering totally different machine types on subsequent replacement orders.

*Clippinger, R. F., "International Standards," Proceedings of the 1961 EJCC, Unpublished, December 1961.

The problems of reprogramming and computer conversion can be minimized by good documentation and sensible machine selection. But the more basic issue is: Will POLs encourage good documentation? The pro arguments hinge on the following points: 1) POLs are "readable" and hence the documentation is "built-in"; and, 2) the documentation will be consolidated at the POL language level, thus eliminating intermediate forms of documentation and making it easier to keep the documentation up to date.

Unfortunately the proponents don't discuss all the documentation. Any well-documented production job consists of:

1) Narrative description of problem
2) Flow charts
3) Annotated code
4) Users' input instructions (keypunch instructions, etc.)
5) Output format and description for user
6) Operators' instructions (including restart and recovery procedures)
7) Tape status log and associated proof of file validity
8) A narrative functional description for the user.

For a long-lived production job of significance, all of the above are required. A POL can only assist in decreasing the effort expended on obtaining annotated code, but even this is dependent on management enforcement. To keep the POL statements "readable" will require discipline and training of programmers--with, as usual, the more experienced programmer being the prime offender because of his desire to use shorthand descriptions and narratives for the sake of his own efficiency and that of the compiler. Documentation will be consolidated at the POL level only if this is the most efficient form for the programmer. If he discovers that some intermediate form is more efficient because of long compile times, difficulty in debugging at POL level, etc., then the advantage will disappear.

Program Checkout (Debugging)

Since one of the major items in the programming process is program checkout, it was natural that almost immediately claims would be made that POLs result in great improvements in debugging computer programs. The arguments supporting this claim went this way: Since a programmer has to write down substantially fewer symbols when writing in a POL (and there is probably a relationship between the number of errors and the number of symbols written), there should be fewer errors in the routine. However, there is a counter-balancing factor to this claim. Most POLs existing today have at best marginal debugging aids, and even worse, almost all of them require debugging at the machine language or symbolic language level--complicating both finding the errors _and_ fixing them.

To avoid excessive cost in machine time, present practice dictates that an object deck be taken as output from the compiling process. System checkout is performed using this object deck. Thus, the problem statement exists in two forms: the source deck and the object deck. Debugging is done in the object language, which is patched several times between compilations. Frequent compilations are still necessary to clean up the object code. The requirements for bookkeeping are still doubled, since both the source deck and the object deck must be maintained during checkout.

At the present stage of development, a higher level language and its associated compiler offer no _great_ advantages in reducing either the elapsed time or the total effort in checking out a program of significance. It should also be noted that trainees or junior programmers may have _more_ difficulty in debugging and will often have to depend on a senior programmer for consultation. It is likely, however, that POLs can reduce somewhat the number of debugging runs for senior programmers.

Programming Costs

Another consistent claim for POLs is that they will reduce programming costs. This is at best an ethereal claim since cost data in terms of elapsed time, man-hours, dollars, or any other metric is a major unknown in the computing field. There is reason to believe that POLs used in an established computing group may indeed reduce programming costs or perhaps more correctly, they may shift major portions of the cost of programming to the machine. Since hardware costs are dropping, this may be a reasonable tradeoff. However, when POLs are used by groups which are composed largely of either inexperienced programmers or "open shoppers," then experience indicates that total programming costs will probably rise. This is largely due to the fact that inexperienced programmers using complicated POLs consume huge quantities of machine time--the catastrophic goof, which results in many compilations, debug runs, and generally (the most common symptom) excessive printed output, is quite common.

Since programming cost is the least understood area in computing, it is probably unwise to attempt to justify or reject a POL on these grounds. The only approach at the present time seems to be the "scarce resource" approach, as summarized from a seminar held during this study:

> The manager's prime objective is to achieve the most efficient use of his resources, the prime two being programmers and machines. Since software essentially enables one to control the tradeoffs between the two, the manager should attempt to adjust the tradeoff in favor of the scarce resource.

Better Understanding by Higher Management

Perhaps the most flagrantly overstated claim made for POLs is that they result in better understanding of the programming operation by higher-level management. It is barely true for the experienced computer facility manager who, because of a POL, attempts to establish more accurately factors such as job mix, resource allocation, programming costs, documentation procedures and costs, etc. In any case, the probability that any level of management above the facility manager will understand programming is independent of the introduction of a POL. The fact that some of the POLs have a narrative quality and program listings which can almost be "read" has only the smallest possible effect on higher management. Built-in documentation will help if it is indeed built-in, but the probability of higher management being concerned with annotated code is indeed slight.

In summary, then, it can be seen that most of the claims for POLs represent not clear gains but tradeoffs made between various resources. The development of higher-level languages is another step (and perhaps as we gain experience, a very significant one) in the evolution of the programming field but it does not seem to constitute a breakthrough.

PROGRAMMING LANGUAGE SELECTION

The problem of selecting an appropriate programming language is a complex one, involving closely-coupled management and design considerations. Management's evaluation of the basic nature of its task should strongly influence the design characteristics of the POL and, by the same token, fundamental design limitations should influence the mode and range of application of the POL.

Management Considerations

The manager of a computing group must consider the basic characteristics of his task before he chooses a POL. Whether his task is to manage a computing facility--job shop operation--or a large-scale application--developing a Command and Control system--will influence the emphasis and importance given these characteristics.* Characteristics common to both types of efforts are:

1) Prime function of the computing group--Is it providing computing services for an R and D group? For company administration? Or is it a major part of a Command and Control design, development, and production team?
2) Resource mix and available tradeoffs--What is the scarce resource--programmer time or machine time?** The manager must consider the type of programmers who use the machine and the amount of control he has over this; e.g., quantity vs. quality, experience level, etc. Is one more concerned with ease of modifying programs or with "tight" object code? Should one minimize elapsed time or machine time? A faster, cheaper machine may require less-efficient software.

---

*The following lists of characteristics do not pretend to be complete or exhaustive--only indicative.

**Most early compilers justify their existence on this basis: successful compilers probably achieve as a guess (unfortunately no statistics) a tradeoff of tripling or quadrupling the instructions per man-hour at the cost of doubling the machine time used. This tradeoff is reasonable under a number of conditions: 1) to increase throughput for constant number of people or, 2) to permit inexperienced users to utilize the computer (open shop).

Facility Manager. Characteristics of concern to the facility manager are:

1) The range of problems; i.e., business or scientific, mathematical or data processing.
2) Open vs. closed shop.
3) Problem turnaround time; i.e., many short problems (short-lived) or a few large problems (long-lived, many changes).
4) The allocation of programmers' time; i.e., a code checking environment or a highly production-oriented shop; do programmers spend more time in formulation and analysis or in coding and debugging?

Application Manager. Characteristics of concern to the application manager are:

1) Environmental constraints--military operational system, real-time system, peacetime logistic system, etc.
2) Design and implementation approach--integrated system, automation of a single function, a new automated function, etc.
3) Resource constraints--tight budget, crash deadline, existing hardware, etc.

In summary, all of these characteristics bear on the decision to use or not use a POL, and how to select the "best" POL for a particular application or computing installation. The proper choice of software essentially enables one to emphasize desired functions and to select tradeoffs between resources. Additional considerations revolve around the design limitations of current POLs.

Design Considerations

The prospective POL user should be extremely careful when claims are made for a universal compiling system.

Languages and compilers for narrow application areas are now indeed feasible and economically practical, even when machines of several manufacturers are involved, _providing_ the machines are of the same basic capability. On the other hand, the design of a language and compiling system which will be all things to all men, in all application areas, and on all machines of all manufacturers is, at this time, a totally unfeasible economic action.

Design Limitations. When a language is designed, the design team must have some application in mind. As a POL is implemented, the programming staff makes use of its accumulated knowledge and the particular characteristics of the source computer. Every POL is a compromise venture--in particular, the compiler portion of a POL is subject to even more compromises. Naturally, any well-designed compiler exploits the characteristics of the object machine. The net result is that all POLs have design points, but it is the compiler portion of the POL that sharply defines the design point. For example, one might standardize on the communication language portion of a POL (which seems to cover a broad spectrum of applications) only to discover that no one can design a _single_ compiler that _efficiently_ covers this spectrum. Thus each "language-source computer-compiler-object computer" combination has a design point. Even if a hardware/software combination is used at its design point, there are good and bad designs. Intuitively, one would guess that a hardware/software combination must suffer materially if used for applications very far from the design point application. The most simple of exercises proved that this was indeed true. The JOVIAL compiler was used to produce code for simple algebraic manipulation. In producing this code, its compile times were materially longer than those for FORTRAN and the object code was significantly less efficient.* Thus, when the JOVIAL compiler was given applications which were precisely on the design point of the FORTRAN system, serious deficiencies resulted. On the other hand, the JOVIAL compiler will handily treat partial-word, scaled arithmetic, using packed fields--something the FORTRAN system will not tolerate at all.

As further extension of this reasoning, the FORTRAN compiler, designed with a large magnetic tape oriented machine in mind, is so unwieldy and has such excessive compile times when used on a smaller machine (such as an IBM paper tape 1620) that its usefulness is indeed questionable. What is at stake in POLs is not just feasibility ("Can I do it at all?") but also efficiency ("Can I do it at reasonable or lower cost?").

Even when compilers are used to produce programs which are at or near the design point combination of application, hardware, and software, the object code produced is still usually less efficient than a code tailored to the particular conditions by an "expert" programmer. This lack of efficiency is apparent both in the execution time of the object code and in the storage required to accommodate that code. One point of encouragement here: many applications do not require the services of an expert programmer. (This is indeed fortunate since expert programmers are in short supply and inadequate efforts are being expended to increase the supply.) Therefore, in the case where an application does not press the limits of the installed hardware, the inefficiencies in the object code are more than outweighed by the advantages of allowing a junior programmer to produce useful results. On the other hand, in the case when the application does indeed challenge the capacity of the installed hardware,

---

*See Table 1.

TABLE 1

COMPARISON OF COMPILE AND EXECUTE TIMES

	JOVIAL		FORTRAN		RATIO JOVIAL:FORTRAN	
					COMPILE TIME	EXECUTE TIME
	MIN	SEC	MIN	SEC		
COMPILE 200 ARITH-METIC STATEMENTS	1	50		45	2.4:1	1:1
COMPILE 150 LOOPS, 1 DIMENSION	2	33	2	0	1.3:1	1:1
COMPILE 75 LOOPS, 2 DIMENSIONS	3	5	1	30	2.0:1	2.9:1
COMPILE 50 LOOPS, 3 DIMENSIONS	2	45	1	20	2.0:1	4.3:1

the inefficiencies in object code produced by present-day compilers may be intolerable.

In the former case, the services of the junior programmer are augmented by the use of a higher-level language, a compiler, and a machine on which to compile. His efforts are amplified and he is able to accomplish an otherwise unmanageable task. _No claim is made for the economic justification of this type of operation_. The only notice which is given is that this is _a_ way to accomplish the work with the resources at hand.

In the latter case, a staff of senior programmers will be required because they must know the language, the compiler, the source machine, the object machine, the problem to be solved, and, additionally, must have an adequate debugging technique on the object computer. This combination of skills is rare and shows all signs of becoming even more rare unless steps are taken to reverse the trend. Under the conditions indicated above (where the application taxes the installed computer capacity) these specialists will, of necessity, resort to hand polishing the compiled code in order to achieve the requisite efficiency. The operational program for a real-time Command and Control system almost always falls into this latter category.

Design Tradeoffs. It is important to discuss in detail some of the significant design tradeoffs in order to sharply describe the current state-of-the-art. These tradeoffs are:

1) _Efficient human communication language vs. efficient compiler source language_.

In a larger context this tradeoff is the result of the man-machine communication problem. The way humans use, process, and transmit information is sufficiently different from the way we use computers to manipulate information that a communication language design which is optimized for humans is generally inappropriate (or at best, very inefficient) for use as a source language to compilers. Currently POL's are forced to push this tradeoff in the direction of efficient source languages and hence they are "readable" to humans only in the most narrow sense.

2) Expressive or "rich" source language vs. compile time.

The logical extension of the above tradeoff is reflected in the tradeoff between expressive or sophisticated source languages and compile time. In general, the narrower the range of the source language, the shorter the compile time. This is the trap that awaits the developers of a communication language such as ALGOL. In attempting to incorporate every possible suggested feature, they are imposing increasingly severe design requirements on potential ALGOL compilers. This inevitably leads to "restricted" compilers, such as "basic ALGOL," "SMALGOL," etc., which delete features in the source language to acheive reasonably efficient compilers.

A similar tradeoff exists between sophisticated source language and effort in debugging. The more sophisticated the language, the more likely it is that one will make mistakes or use a feature of the language improperly. This results in more debugging runs to discover these errors, or complicated "scanners" in the compiler to discover language usage errors. Since sophisticated source languages require sophisticated compilers, it becomes increasingly difficult to determine why the compiler generates certain segments of object code in a particular fashion. This aspect is further complicated by the fact that most debugging today is done in object language rather than source language, which makes the debugging process that much more difficult and involved because two different levels of language are being used.

3) Compile time vs. object code efficiency.

Finally, one of the most important tradeoffs in compiler design is between compile time and object code efficiency. Object code efficiency generally is proportional to the length of compile time. So-called "load and go" compilers achieve this capability at the expense of object code efficiency (or by severely limiting the richness of the source language). The opposite approach is to utilize compilers with elaborate optimizing routines to reach more acceptable object code efficiency. This, of course, results in long--and to many users totally unacceptable--compile times.

Cost of Compilers. Present-day compilers are expensive to produce and expensive to maintain. The cost of producing a modern compiler, checking it out, documenting it, and seeing it through initial field use, easily exceeds $500,000. Once the compiler is delivered to the field installation, its costs continue in the form of maintenance and improvement. These continuing costs are at least two man-years ($40,000) and approximately an equivalent amount of machine time costs for the use of the compiler maintainers.* This additional cost continues for the entire life of the compiling system. These costs are usually not openly discussed when the praises of a higher level language are being sung.

*From time to time, competent programmers question the above figures and cite examples of the development of low-cost compilers. Invariably, these are cases where the compiler was developed for in-house use only. As soon as the effort involves several outside users in diverse geographic locations, costs begin to rise rapidly as the increased need for documentation (e.g., training manuals, narratives, and descriptions for the user), maintenance and revision procedures, user training, additional special user requirements, etc., become apparent.

If the compiler is being written for a computer which does not yet exist, and which does not have a basic set of checked-out software, and if elapsed time is an absolute premium, the cost can be double or triple the above amount.

Compilers, and the shortage of senior programmers, amplify a growing difficulty in the area of hardware maintenance. While computers have become more reliable in the past several years, maintenance personnel have become less qualified. The hardware maintenance area is suffering from the same shortage of senior men as the programming area, and the situation will get materially worse unless steps are taken to arrest this undesirable trend. In the past, occasional serious difficulties would occur which would not show up when the standard hardware diagnostic programs were run. To solve this dilemma, the most hardware-oriented programmer and the most software-oriented maintenance engineer would sit down, discuss their mutual problem, and proceed to map out a series of experiments which would confirm or deny their suspicions. This usually required carefully written diagnostics to be produced on the spot to the specifications agreed upon. Unfortunately, higher level languages and their compliers divorce the inexperienced programmer from the computer hardware. Thus, there is a smaller population from which to glean senior hardware-oriented programmers. Simultaneously, the accelerated production of computers has increased the requirement for personnel in this unique category.

This team effort is rapidly vanishing in the field simply because the competence is not there. In a civilian environment, a deficient piece of hardware merely raises the number of reruns and the associated costs until it is located and repaired. However, in a military environment, backup machines (or excess machine time) may not be conveniently available. Furthermore, should such a malfunction occur during an emergency situation, the entire computer-based system could break down at precisely the time when it is needed most.

MEASURES OF PERFORMANCE

In a young rapidly growing field major advances come so quickly or are so obvious that instituting a measurement program is probably a waste of time. At some point, however, as a field matures, the costs of a major advance become significant. The problem of choice appears since now several (or many) alternatives are feasible, and we become more sharply aware of the tradeoffs required to achieve a gain in performance. When the POL arrived on the scene it seemed to offer gains in several areas--built-in documentation, reduced training, etc., and more efficient use of programmers' time. However, the magnitude of the gains is questionable, the total costs (in the larger sense) are unclear, and comparative studies of POLs are rare and equivocal. Thus, a measurement program is needed now to more accurately assess the gains; develop costs in machine time, direct, and indirect support; and to institute a sound program of comparative studies of POLs.

Deficiencies in the Current Program

In measuring POLs, several performance criteria are likely to be relevant. For a few users a single criterion will suffice; for most users, some intuitive mix of criteria will be required. As is often the case, choosing these criteria is difficult because the requirements and operating procedures which are appropriate for one criterion often conflict with those of another criterion. For example, one may wish to "maximize throughput/dollar" (this implies batch processing; i.e., stacking jobs in a queue) and minimize "turnaround time" (this implies quick access to the machine; i.e., short or non-existent queues).

In the development of a large data processing system, a manager may wish to minimize a project's elapsed time, which will conflict with criteria such as "minimum cost" and "maximum throughput/dollar." Thus, any measurement program will require: 1) the definition of relevant criteria, 2) the appropriate selection of criteria, and 3) the development of measurable parameters which relate to these criteria. The first two are management decisions. The third sets a firm requirement for a broad measurement program to develop these parameters.

Measurement Program Outline

A sound measurement program always requires an extensive data collection effort to establish accurately the variegated costs, to establish bases for comparisons, and to determine the tradeoffs involved. A measurement program should include at least the following factors:

1) Machine Time--The allocation of machine time into the following categories:
   a. Overhead--Machine time spent in compiling or assembling routines (plus a breakout of the duty cycle of the component programs in the compiler or assembler); the amount of time spent on compiler or assembler maintenance and improvement; and finally, the amount of time spent in the non-execute phase of programs--loading the program, system moves and transfers, idle time, etc.
   b. Checkout and Shakedown--Breakout of time spent running the program non-productively; i.e., checking it out, running test cases, etc.
   c. Production--Time spent on production runs. It is also important to know the percentage of time spent in the input-output phases and

in the computing phase. This should be a strong factor in software design and hardware configuration selection.

d. Maintenance--Time spent in either preventive maintenance or fault-locating on the machine.

2) Programmer Time--The allocation of programmers' time in the previously defined phases of programming is a basic requirement. The man-hours spent in problem formulation, coding, checkout, final documentation, and overhead (training, reading manuals, etc.) would constitute a minimum list of factors.

3) Dollar Costs--A diligent effort to establish dollar costs of all phases of the operation of the computing installation is essential. Such factors as machine rental costs (including off-line equipment), support equipment (key punches, verifiers, etc.), programmer costs, support personnel (operators, etc.), software system costs (initial plus on-going), are essentials to any measurement program.

Performance Parameters

The most glaring deficiency in the software area is in performance parameters. This deficiency will remain until we develop the cost and data collection endeavor outlined above and rigorously define each process and subprocess in the programming area. In the absence of these definitions, already complicated interrelationships become indescribable. We must be able, at some point, to analyze multiple criteria and complex performance tradeoffs. For instance, since POL compilers cost substantial sums (initial investment plus continuing cost) their cost must be factored into the dollar costs of machine and programmer time which in turn must be factored even further (see listing above). Even if an existing compiler is used, the continuing cost of modification and "mothering" of the compiler cannot be neglected. The cost (in machine time) of a compilation is significantly more than of an assembly. Factors of four to ten are not unusual; hence they represent a significant cost factor.

A further penalty paid because of the lack of defined performance parameters is the inability to compare two POL systems on any basis except a subjective, qualitative one. As one author of a current comparative evaluation put it:

> Language design is still as much an art as it is a science. The evaluation of programming languages is therefore much akin to art criticism--and as questionable.

Finally, since the definition of what constitutes the makeup of any given language (and compiler) is constantly changing, it is impossible to repeat any given test or evaluation without obtaining markedly different results. This naturally adversely affects both language design evaluation and comparability.

There are three areas of POL work in which the measures of performance would materially advance and clarify the efforts invested. First is the area of design of POLs--the ability to use and manipulate performance parameters in new and proposed designs would prove invaluable. Second, measures of performance would assist in estimating efforts to produce a new POL. Currently our estimating ability is pure crystal ball--we lack sufficient parameters to require even the back of an envelope. Finally, decision-making regarding proposed modifications or revisions could be put on a quantitative basis. The current approach ranges from qualitative assessment to sheer guesswork.

Three Measurement Studies

Listed in the Appendix are the results and conclusions of three studies which support much of the discussion in this Memorandum. These three studies suffer from: 1) limited range of the tests; 2) terms, ground rules, and test problems that are poorly or totally undefined; and 3) lack of cost data in comparable terms. Nevertheless, the reader is urged to peruse these studies since the results do convey some knowledge and the identification of the deficiencies in each study is instructive. Further, they indicate clearly the problems of test design and evaluation which any measurement program faces. It should also be stressed that in spite of the fact that these results must be treated gingerly, efforts of this type must be encouraged (and their limitations understood) rather than attempting to discover what constitutes optimum test design. Incidentally, these three studies represent about a third of the known studies.

SOME DESIRED COMPILER DEVELOPMENTS

There are several compiler developments which should be encouraged and supported. From successful developments of the type indicated below, it would be possible to obtain compiling systems which have broader applicability, lower costs (both first time and continuing), and increased efficiency (any definition).

1) Modularity - Some development efforts are proceeding whereby truly modular compilers can be realized. If this can be accomplished, the user may have the option of adding or removing editors, debugging routines, code-polishing routines, or partial-word arithmetic. Thus, major sections of code can be deleted from the compiler. The resultant operational compiler will be better

suited to the purpose and, since it does not carry as large an overhead burden, will be faster and cheaper to operate.

2) Adaptability - Compilers with flexible internal structures are also just appearing. If these developments can be followed to their logical conclusions, it may be possible to have compilers which, for example, generate either fast object code or compact object code. Thus, a programmer may choose the tradeoff suitable for his application.

3) Debugging - In the past, the area of debugging has been almost completely ignored. Most compiler writers assumed some utopian individual who did not make mistakes when writing source language. Those designers who did not adhere to that contention believed there was no cost associated with machine time, and lengthy compile times could be tolerated. Finally, other compiler designers believed that they could determine all of the errors in object code if only they had adequate source language editing. Unfortunately, all three of these contentions have proved to be false. The debugging area must be attacked in its own right to achieve the oft-advertised but elusive benefits of one language level. Until one language level is achieved both for describing the process and debugging the code, training requirements will be increased and costs will be excessive.

4) Range of Application - Some efforts are now under way to broaden the range of application of compilers. Several compiling systems will allow the intermixing of one or more source languages

in a single program. Unfortunately this is usually accomplished not by merging the languages but by allowing mechanical intermixture, a technique which yields tougher debugging and more errors. On the other hand, there are one or two splinter efforts aimed at trying to find one basic language which will be adequate for formula evaluation, data processing, and real-time control. Although this may not be possible in the immediate future, such efforts should be encouraged.

5) Integrated Design - Some manufacturers are organized internally so that hardware is the result of one design group and software is the result of a second independent design group. If performance measures such as compile speed, expansion ratio, and object efficiency (all three now undefined) are adopted, these two design groups will be forced to merge their design efforts in order to produce an integrated and balanced package. Thus, hopefully, the phenomenon of the compiler which is designed to overcome deficiencies in the hardware design will gradually fade from view.

SUMMARY

There are several key points about POLs which should be summarized:

1) Tough programming jobs still require top-flight people (use any intuitive notion of what is meant by "tough"--tight real-time constraints, big jobs, short deadlines, etc.).

2) POLs make a contribution in only part of the programming process. Equally important facets that need significant contributions are:

   a. System design and problem formulation--the development of data processing system parameters, and of methods and techniques for describing data processing functions.
   b. Data organization and mapping--the establishment, care, and feeding of a data base; the problems of defining, formatting, and error-checking data.
   c. Compatibility--POLs are only a partial answer at best.

3) No single current POL can <u>efficiently</u> cover any substantial portion of the computer applications spectrum. This may change over time--but slowly. Therefore, only count on a POL to cover a limited set of applications.

4) There are alternatives other than embracing a POL:
   a. Communication language--a communication or specification language will often bring most of the benefits of a POL without the large investment.
   b. Macro-oriented languages--these offer many of the POL advances and are appropriate where inefficient use of the computer cannot be tolerated.

III. THE STANDARDIZATION QUESTION

BACKGROUND

In examining the question of standardizing on a POL, it is appropriate to discuss the broad question of standardization. DOD Manual M-200 lists as the purpose of standardization:

1) to improve efficiency and effectiveness of a function and,
2) conserve money, manpower, time, production facilities and natural resources.

Usual objectives of a standardization plan are to minimize the number of items; optimize interchangeability; standardize terminology, codes, and drawings; etc. Further, the implied uniformity of a standard allows easier and more accurate estimation of both capabilities and costs, improved communication among users, better and more accurate statistics, and more readily achieved compatibility. In addition, a standard will allow all affected agencies to concentrate their energies and resources on using and/or improving the standard.

The advantages and needs for standardization are well recognized in the United States. Interestingly enough, most of the standards in the U.S. are set by various technical societies. There are over 350 organizations in the U.S. involved in standardization activities. In Standardization Activities in the U.S., Sherman Booth says:

> The national technical societies of the U.S.A. are the very backbone of its standardization achievements. This fact sets our country apart from others wherein the results of standardization stem from a mandatory rather than a voluntary basis.

Given the advantages of standardization, how do we know when to standardize? There are certain precursors which indicate a readiness for standardization: a sufficient base of consistent (or standard) terminology and accepted lower-level standards on which to build; a relatively mature state-of-the-art; and extensive user experience. However, one should not fall into the trap of believing that the rate of progress must approach zero before a standard can be instituted. Booth points out:

> Standards are not static. As the rudiments of technological aspects of problems become commonplace, as greater knowledge of the chemical and physical characteristics of products are more widely known and accepted, a committee may again be activated to reconsider and modernize or improve a previously issued standard. Such improvements are regular, frequent, almost routine. There are relatively few documented standards that have never been revised. Thus, standardization agencies find themselves engrossed in the problems of revising standards as well as developing new standards, and to about the same degree. Standardization is dynamic. It must necessarily follow closely upon the heels of science, research, invention, and creation if it is to serve its intended purpose.

We will attempt, in the following discussion, to indicate our views of the level of standardization in the programming field that is achievable in the next few years.

DEFINITIONS

Before beginning our discussion, a few definitions will ensure a common basis of understanding.

STANDARD	(Noun) That which is established by authority as a rule for the measure of quantity, weight, quality, etc. That which is established as a model or criterion to measure against.
STANDARDIZE	Applying political and/or economic sanctions to enforce usage of a standard. Standardization is appropriate when, in the

	eyes of an authority, the benefits of further experimentation are exceeded by the benefits of standardization. In standardizing, the authority must define (or describe) the standard and how to verify it. By standardizing, all efforts are brought into line and further progress and experimentation can proceed from a uniform base.
STANDARD	(Adjective) Used in conjunction with phrases such as "programming language." The resulting phrase implies a language that has been subjected to heavy use and experimentation and found to be the most suitable for its field of application either by a central authority's evaluation or by widespread usage.
CHARACTER SET	That collection of basic symbols and signs which is used to depict meanings. English speaking countries usually have, as a minimum, the ten symbols 0 through 9, and the twenty-six symbols A through Z. A heated debate is now raging regarding the total *number* of characters in the set used in the computing field (choices frequently mentioned are 48, 64, and 128); and what additional *symbols* will be added to the set to make up the total (special characters such as >, <, $, %, etc.).
COLLATING SEQUENCE	Given *any* character set, a design decision must be made on how to order these symbols within the computer. Modern computers contain a collating sequence implied in the internal wiring of the device. The compare order, the sorting function, the printer circuitry, and the card read circuitry

are all severely affected by the <u>arrangement</u> of the chosen characters into an ordered sequence. The discussion centers around the placement of the aforementioned special characters and whether numbers take precedence over the alphabet or conversely.

REQUIREMENTS FOR A STANDARD POL

As we have indicated earlier, there are at least three requirements that a candidate for standardization should fulfill: 1) a sufficient base of lower-level standards, 2) a mature technical development, and 3) extensive user experience.

A Sufficient Base

The base on which a standard POL rests should consist of key basic standards and standard terminology. An essential requirement is standardization at lower levels, such as collating sequences, card and tape formats, character sets, etc. Unless we first standardize at these lower levels, any higher standard will be apparent rather than real. That is, if <u>all</u> incompatibilities must be resolved at the POL level, the impact of this requirement will necessitate numerous versions of the standard POL and the single standard will be a fiction. Moreover, without a standard terminology, we cannot even describe and discuss a standard POL. Much <u>basic</u> work must be accomplished--work which even if attacked vigorously will probably require a minimum of two years. The pace of standardization in programming languages will be set by our progress in attaining this base.

Mature Technical Development

A second requirement is for mature technical development. An important indicator of this development will be the emergence of POLs with sufficient scope to efficiently cover large and significant application areas and a wide class of machines. Progress is being made, but much improvement in performance is still required.

Another indicator of progress in technical development will be the establishment of standardized measures of performance. It is encouraging to note that in the last few months, more and more attempts at <u>quantitative</u> measures of POL performance have appeared. This is a most healthy trend. However, all of the studies to date require extensive interpretation; significant conclusions can be drawn only at some substantial risk. Measurement efforts should be accelerated and encouraged but will bring real clarification and guidance to the field only if standardized performance parameters are defined and developed. We must, over the short haul, factor out those aspects of the programming problem which we cannot currently measure and concentrate on those we can. Advances in this area are absolutely essential to the selection of a standard POL. Any sensible standardization plan must provide for standard acceptance tests and a method of evaluating, and ranking in priority order proposed revisions to the standard. Most of the value of a standard resides in its slowly changing nature and the guarantee that it performs as specified.

A third indicator of technical maturity will be the development of an adaptable and modular compiler for the standard POL. Even if a communication language can be developed for a wide application area and its associated compiler can cover a large class of machines, it should be recognized that each computing application and facility has unique requirements and constraints. The capability to "tailor" the compiler to the particular needs of each major application and facility is a key technical requirement for POL standardization. Important advances in the development of modular compilers and the development of compiler design criteria will be required.

Extensive User Experience

Extensive user experience and wide acceptance are simply healthy indications that the POL has been subjected to a free market and is fulfilling the needs of a wide spectrum of users. In reviewing the state-of-the-art in Sec. II, we saw that most current POLs fail to meet this necessary condition.

PLANNING FOR STANDARDIZATION

Much of the preceding discussion has emphasized the problem of assessing the need for standardization and the problem of selecting a standard. Equally important are the problems of maintaining and improving a standard. These are primarily problems of efficient organization and resource management--new organizations will be required, new procedures, quality people of a type already in short supply, and, of course, money. Due to the magnitude of the investment, it is mandatory that considerable advance planning precede any serious step toward large scale standardization. Since the Air Force is one of the largest users of computers in the U.S., a central Air Force agency responsible for the planning and implementation of a standards program is absolutely essential. The Air Force Directorate of Data Automation would seem to be the appropriate agency.

In addition to the management side of the standardization effort, the maintenance and improvement effort will require significant technical efforts to examine such fundamental questions as the tradeoff between maintaining a relatively static standard and incorporating improvements in the standard. In this tradeoff, the rate of innovation will slow over the present rate but at the same time we should develop sharper and clearer ideas of what are high payoff improvements. For example, this should result in improved specifications for the Air Force applied research program in computer techniques.

PARTIAL STANDARDIZATION

There are several possible alternatives short of complete standardization. One alternative is to adopt a local standard which centers on a single application area _and_ a single machine type. The application area must be carefully and narrowly defined to keep it within an efficient design range of current POLs. Command and Control is clearly far too broad an application area. More typical applications might be headquarters level intelligence, base level logistics, etc.

A second alternative is to standardize on one component of the POL--the communication language component. This would allow a broader application spread (since a compiler is not involved), but would have to be tempered by the number of compilers required to cover the application area.

Both of these approaches offer modest but important advances and retain flexibility for full standardization when the art permits. They are conservative approaches and are well within the current state-of-the-art.

SUMMARY

On the basis of this study the answer to the question of programming language standardization is: not now. Key and significant requirements for standardization are unfulfilled. However, the prognosis for standardization in the relatively near future (2-4 years) is good _if_ the following tasks are carried out:

1. Work on basic standards is required as a prerequisite to POL standardization. Standard terminology and measures of

performance are prerequisites to standardization. A standard can be declared by fiat, but without these factors its value is at best questionable.

2. Programming languages with increased range of application need to be developed. The current POL state-of-the-art is still not up to the requirements of complete standardization.
3. Planning should be initiated now on the organizational, resource, and technical requirements for standardization--preferably under a central Air Force agency such as the Air Force Directorate of Data Automation.
4. Partial standardization options should be considered:
   a. Local standardization.
   b. Communication language standardization.

IV. CONCLUSIONS

The foregoing discussions have defined some terms, stated some facts, and discussed both sides of topics concerned with programming problems, the current state of the programming language art, and the standardization question. The following conclusions are set down in a convenient order, with no ranking or implied importance to be inferred by the order of presentation.

1. Standardization

In specific answer to the question, "When should the Air Force standardize on a common programming language for Command and Control?" the study indicates: NOT NOW. This is the only conclusion that could be reached since: a) so many important facts remain unknown (not necessarily unknowable), b) the state-of-the-art in programming languages exhibits considerable immaturity, and c) much basic work in standardization remains to be done.

2. Basic Standards

The Air Force should lay the ground work and prepare to take a stand on some basic standards in the computer field. Efforts are now under way in the American Standards Association (ASA) to draw up a number of representative standards and submit them for review. Certain basic standards are long overdue. Action is recommended for standardizing on:

a) A data processing glossary.
b) 80-column cards.
c) Magnetic tape recording formats.
d) Flow chart symbols.
e) Character sets and collating sequences.

A glossary is long overdue in the computer field. This lack of a glossary is impeding progress and placing an additional burden on military officers who must become conversant with the field in a minimum period of time. To further complicate matters, the several manufacturers make overt attempts to avoid using the same word to describe the same hardware feature.* It is recommended that the Air Force let it be known that it will accept the ASA glossary when it is published. Furthermore, the Air Force should allow some reasonable grace period, 24 months for example, after which it will buy and accept only computer equipment whose manuals and specifications are written using the terms as defined in the published ASA glossary.

The 90-column punched card has been declining in usage for some time. Its demise was sealed when its principal proponent elected not to offer 90-column equipment as standard input/output for part of his new product line. Recognizing that a clear trend has been established, the Air Force should curtail the installation of any additional 90-column card equipment unless extenuating circumstances so dictate. Furthermore, the Air Force should have a workable plan for replacing 90-column card equipment with 80-column card equipment on a gradual phase-out basis.

A similar opportunity for establishing a basic standard exists with magnetic tape drives. A distinct trend is already evident among manufacturers toward half-inch tapes, recorded with seven tracks at either 200 or 556 bits per inch, with a three-quarter inch gap between records. This is the level at which standardization is feasible and practical in the immediate future. It is typical of the level of standardization which must precede standardization at the programming language level and has the added benefit of immediate payoff. A standard tape format would greatly benefit the exchange of data files between installations.

---

*These attempts have gone so far that two manufacturers cannot even agree upon the spelling of disk or disc.

If this standard were adopted for communication purposes between machines, there would yet be many avenues open whereby additional tape performance could be obtained. Thus the development of newer and better equipment would not suffer materially.

There are several excellent sets of flow chart symbols and conventions available at the present time. A basic set could be chosen and would result in increased efficiency and decreased costs to all concerned.

The subject of character sets and collating sequences is also long overdue for standardization. The Department of Defense exhibits one of the principal instances of an in-house incompatibility. The Fieldata character set which has been adopted in the communications area is incompatible with most of the electronic data processing equipment the government has presently installed. Until the Department of Defense takes positive action and reports its stand to industry, the confusion is likely to continue. Again, the ASA is working in this area. Should the Department of Defense accept what the ASA recommends, a large forward step would again be taken. The standard should not be adopted overnight, but only after a period, such as four years, to allow the replacement (through attrition) of print drums and code wheels on input/output devices.

If tape formats, character sets, and collating sequences were commonly accepted, the military would have come a long way towards being able to interchange files of data between interested installations. This area, while not as romantic as higher-level languages and compiler programs, would begin to return dividends immediately and requires no further research or development.

3. Partial Standardization

The study indicated that the Air Force should go slow in the adoption of compilers to be applied to a broad spectrum of problems. There appear to be no dangers (other than the costs indicated in the previous sections) in the adoption of languages and compilers where a single application area <u>and</u> a single machine class are involved. This might be considered adopting a language on a local scale. On the other hand, there appear to be grave inefficiencies if present-day compilers are adopted on a global scale.

Another alternative to complete standardization is to standardize on the communication language component of a programming language--letting each user select the type of processor (compiler, assembler, etc.) appropriate to his particular requirements.

4. Central Agency

As indicated in this Memorandum, the Air Force urgently needs some central agency charged with responsibility for:

a) <u>Measures of Performance</u>--An effort is needed to gather, keep, and organize statistical studies on an unclassified, objective basis. The lack of facts, performance parameters, and adequate measuring techniques must be overcome if we wish to quantify our decisions. This effort probably must go hand in hand with the establishment of a glossary, for, in the interim, word meanings cannot be established with sufficient rigor.

b) <u>Planning for Standardization</u>--The Air Force's large investment in computers and supporting data processing equipment requires that a careful and continuing planning effort be permanently established. Good (and bad) decisions on standardization in the Air Force can produce significant changes in the costs of Air Force data processing systems.

In addition, many of the problems of incompatibility between systems and the problems of data, file, and computer program exchange can be resolved as part of an overall standardization and equipment selection plan.

c) Computer Selection--The Air Force needs to use more central planning and deliberation in choosing new computers. Much of the problem of incompatibility (which a machine-independent higher-level language must solve) is often due to the decentralized way computers are purchased by the Air Force. For instance, the purchase of two computers from different manufacturers which are to be applied to precisely the same problem area, though geographically removed, can be avoided by centralized computer selection.

In addition, if elapsed time is a critical factor, the Defense Department is well advised to avoid "paper" or "one-of-a-kind" computers and instead to pick an established computer with checked-out software.

5. Assessing Programming Languages

Programming languages represent an attack on the "programming problem," but only on a portion of it--and not a very substantial portion. Much of the "programming problem" centers on the lack of well-trained experienced people--a lack not overcome by the use of a POL. The problem of training and acquiring top-flight people is not alleviated by the introduction of POLs.

The current state of technical development of POLs is relatively immature. No single POL can efficiently cover any substantial portion of the computer application spectrum. The restricted range of current POLs is primarily due to limitations in compilers. No design parameters have been isolated, nor have the techniques to use such parameters been developed. Several compiler developments would materially assist in widening the design range, increasing productivity, and achieving a resource tradeoff which is under the facility manager's control. Developments in the area of modularity, adaptability, debugging, and integrated design could significantly improve the current picture. The Air Force should take the lead in directing and supporting research efforts in this critical area.

6. Indoctrination and Training Program

Computers and computer programming have developed since most military officers received their basic education. To acquire an awareness and understanding of computer-based systems requires a significant investment in training. It is unfair to expect an officer to acquire the requisite training on his own and hold down a full time assignment. Several other alternatives are available. A short course could be set up to provide the necessary training. Progressive self-study courses could be made available. Roving lecturers could be scheduled for installations where the officer population warrants. These periodic lectures could be supplemented by lists of recommended reading. Audio visual aids could be obtained in quantity. The problem ultimately can be solved by proper academic curricula and appropriate additional courses at the established USAF schools and colleges.

In addition to the problem of indoctrinating the Air Force officer who is a user of computer systems, there is the problem of developing a cadre of skilled, professional programmers in the Air Force. An intensive and well planned training program is required to meet the growing Air Force need, in both quality and quantity, for computer programmers.

A staff study on this critical indoctrination and training program, and how it fits in with Air Force long range objectives in computer-based systems, would materially assist in laying out a course of action and determining how to proceed.

Appendix

THREE MEASUREMENT STUDIES

B. F. GOODRICH COMPANY STUDY

Background - A test designed to compare compiling time, object program running time, COBOL diagnostics, and to prove the operability of the COBOL programming systems. A COBOL source program in RCA 501 format was to be converted to two other computers and comparisons made. The comparison involved three machines: the RCA 501, GE 225, and IBM 1410.

A news release announcing the completion of the study appeared in Electronic News, February 19, 1962. On June 6, B. F. Goodrich Company sent a summary report of the test to those who had requested more details.

Test Problem and Results - The program involved the updating of an inventory file and the printing of a complex inventory shortage report. The COBOL program consisted of 220 data division and 394 procedure division statements. The data available to the programmer consisted of:

1) The COBOL source program in RCA 501 format
2) A flow chart detailing only the COBOL steps
3) A magnetic tape listing of all inputs
4) A sample listing of six pages of the desired output report
5) A brief description of the program.

With regard to the detailed results, we quote from the report: "All of these objectives were achieved in the test; however, it is the decision of the B. F. Goodrich Company not to release the comparative results." The exact reason for this decision is not known. We can suggest however, that perhaps the two losing manufacturers may have taken serious issue with the comparative results and the interpretation of those results.

<u>Major Findings and Conclusions</u>

1) "In one case, a relatively experienced programmer wrote his first COBOL program writing from the COBOL manual and the above data. Two inquiries to the original RCA 501 programmer were made by telephone. Practically no difficulty was encountered that could not be traced to the characteristics of the COBOL compiler under test. About three weeks of part-time work were estimated with much effort being spent in learning and interpreting the two COBOL languages and in creating the extensive test data required. The programmer key punched this data himself. Some key punching errors were made resulting in the only difference between the two operating object programs.

2) "In the second case, a relatively inexperienced programmer who had written COBOL programs before on another computer system attempted the conversion. This person had considerable difficulty in converting the RCA 501 program and, in fact, the services of other programmers were required. The difficulties involved misunderstanding of the RCA 501 source program, mis-use of certain sophisticated features of the COBOL system under test, plus basic misunderstanding of the machine language of the computer under test. In addition, the three magnetic tape input records were created from COBOL programs which "fanned" out the test input data to the desired test size. Thus, a relatively inexperienced programmer was attempting to compile and run four programs on a computer he had never encountered before. Approximately two weeks were required to write the programs. However, a series of compiler or object program difficulties delayed completion of the program for four more weeks.

3) "In addition to two phone contacts with the original RCA 501 programmer, BFG gave some assistance. This assistance was basically "how to program and debug" rather than specific help on the test program. It is important to note that both manufacturers felt that much less time would be required if their programmers had been more familiar with COBOL and the computer under test. In other words, the documentation was adequate, at least for a programmer experienced in how things are done on computers.

4) "Although an inexperienced programmer can write in COBOL, probably COBOL requires significantly higher quality programmers to make use of the full sophistication of the system.

5) "A firm training in the machine language as well as how all peripherals operate is required to produce efficient programs and to facilitate debugging."

STUDY OF PROGRAMMING SYSTEMS FOR THE IBM 650

Background - A test designed to compare several of the programming systems for the IBM 650. The programming systems examined were GAT (General Algebraic Translator), FORTRAN, and SOAP (Symbolic Optimizing Assembly Program), plus "machine language."* In the study, the author rates these languages in order of ease of program preparation:

1) "Writing FORTRAN programs is almost like writing down algebraic expressions.

2) "GAT is similar, but it places more restrictions on the use of symbols.

3) "SOAP is closest to actual 650 machine language, but still a great deal less tedious to produce.

*We chose to ignore the two interpretative systems which were included in this study. "Machine language" in this case is probably a symbolic version of machine language.

4) "Machine language is relatively more difficult to learn and extremely tedious to write."

The study was published in The Behavioral Science Journal, February 1962, entitled, "The Use of Simplified Programming Systems in IBM 650 Data Processing" by Linton C. Freeman, Syracuse University Computing Center.

Test Problem and Results

"The problem selected for study is typical of those confronted in statistical data analysis. It involves the computation of chi-square as a test of significance in a 2X2 contingency table. Such a problem involves a relatively large amount of input and output and employs a straightforward series of arithmetic operations. The logic is simple, and there is no need for any extensive iterative processes."

The results of the study are summarized in Table 2.

Main Findings and Conclusions

1) "The order of the assembly times is roughly the inverse of their programming difficulty."

2) GAT, which had a shorter compile time than FORTRAN, yielded less efficient (in number of solutions/minute) object code.

3) Program read-in time was significantly greater for the two compilers than for SOAP and machine language because FORTRAN and GAT require that standard subroutines be read in along with the program.

4) "Since they require less computer time both for assembly and program running, the machine language systems seem to be called for whenever machine time is at a high premium. Then, too, if a program is to be used over an extended period of time, the greater speed of machine language programs makes them desirable. In other cases, the simplified systems seem satisfactory.

Table 2

COMPARISON OF IBM 650 PROGRAMMING SYSTEMS

Language	Compile or Assembly Times	Time Required to Load Compiled Programs in Computer	Number of chi-square solutions per minute	Number of chi-square solutions per minute [b]
GAT	165 sec.	52 sec.	49 runs/min.	200 runs/min.
FORTRAN	427	33	58	326
SOAP	146	7	100	341
Machine Language	45	5	100[a]	494

[a]Limit imposed by output unit of the computer.

[b]Without read or punch instructions.

5) "If we compare the machine language systems with SOAP, which is considerably easier to use, it becomes apparent that unless the program is going to be used almost all day every day, SOAP will be satisfactory. In a situation where machine time is relatively unavailable, or when a program is to be run quite often, SOAP seems to be the choice among the simplified systems. This is particularly true when there is a great deal of input and output. In such a case SOAP, with its relatively greater read and punch speed, would provide more output.

6) "FORTRAN would be a good choice only when a great number of runs could be anticipated. Its excessive assembly time precludes its use for one-shot programs. Furthermore, its slow rate of input and output suggest that it might best be employed on problems which have little input and output but a relatively large amount of internal computation.

7) "In contrast, GAT assembles and accepts input and produces output relatively quickly, but its rate of computation is slower. This suggests that the appropriate application of GAT is for problems involving a small amount of internal computation, when only one or few runs are anticipated."

BUSHIPS COBOL EVALUATION

Background - The Bureau of Ships, in conjunction with the David Taylor Model Basin, is in the process of evaluating COBOL for shipyard applications. The stated objectives of the project are:

1) "to analyze COBOL capability for handling complex shipyard problems;

2) "to determine the type and utility of the diagnostics;

3) "to determine the extent to which programs can be debugged in the source language;

4) "to evaluate the effectiveness of COBOL for describing business problems;

5) "to obtain information on the use of COBOL to document programs;

6) "to determine the extent to which COBOL must be changed in going from one computer to another."

This was the first of a series of studies to assess various existent COBOL systems. It called for the programming of a Bureau of Ships problem by computer manufacturers who had announced an operative COBOL system. The manufacturers could solve the problem without restraint as to hardware or COBOL configurations if they were available and operative. It is expected that experience gleaned from this initial test will guide the planning of further studies.

The participating manufacturers were Remington Rand, RCA, IBM, GE, and NCR. The first report of the study was published in the Communications of the ACM, May 1962, entitled, "Interim Report on Bureau of Ships COBOL Evaluation Program," by Milton Siegel and Albert E. Smith.

Test Problem and Results - The test problem entailed the development of a "Statement of Operations" Report which depicts the overall financial condition of the eight major activities (cost centers) of the David Taylor Model Basin. The key steps include:

1) A breakdown of current month's expenditures by transaction and expense category.
2) Updating master file to obtain expenditures for fiscal year to date for each

cost center.

3) Printing the "Statement of Operations" Report.

The results of the test are summarized in Table 3.

<u>Major Findings and Conclusions</u>

1) "For this study, no attempt was made to compare the relative effectiveness of the various COBOL compilers because compiling times and running times are strongly dependent upon the method of handling the problem and the ability of the particular programmer assigned to the task.

2) "Gratifying progress has been made by the participating manufacturers in developing COBOL compilers for complex data processing applications.

3) "Standard flow charting procedures are required in order to accrue the full benefits of COBOL.

4) "Continued effort should be made by all manufacturers to reduce compiling time.

5) "Although many programming errors can be found at COBOL source language or intermediate language level, it is almost always necessary to analyze the machine code to completely debug the program.

6) "The object programs produced in the test problem appear to be highly efficient."*

*Personal conversation with one of the authors indicated that this statement was based on Univac II being "rated" twice as fast as Univac I, and since the routines ran about twice as fast, this statement was inferred. It is a purely qualitative judgment.

Table 3

ANALYSIS OF COBOL PROGRAMS

Computer	# of COBOL Statements	# of Machine Instructions	Compile Time (Min.)	Object Program Running Time (Min.)	
UNIVAC II	630	1,950[a]	240	17	
RCA 501	410	2,132	72	12	Used COBOL 60
GE 225	328	4,300[b]	16	12	
IBM 1410	174	968[a]	46	19	Used COBOL 61
NCR 304	453	804	40	NC	

[a]Does not include Input-Output Instructions

[b]GE's results represent a recoding of the problem by an experienced programmer. Original coding (by an inexperienced programmer) was halted when at the end of two of the three required routines his machine instructions numbered 6,419 and compile time equalled 34 min.

NC = Test not complete as of publication date.

BIBLIOGRAPHY

Arden, B., B. Galler, and R. Graham, Michigan Algorithm Decoder, University of Michigan, February 1961.

Armed Forces Supply Support Center, Standardization Division, Standardization Policies, Procedures, and Instructions, Defense Standardization Manual--M200, Washington, D.C., January 1, 1960.

Association for Computing Machinery, "Papers Presented at the ACM Storage Allocation Symposium, June 23-24, 1961," Communications of the ACM, Vol. 4, No. 10, October 1961, p. 416.

Bemer, R. W., "Survey of Coded Character Representation," Communications of the ACM, Vol. 3, No. 12, December 1960, p. 639.

Booth, Sherman, Standardization Activities in the U.S., Government Printing Office, Washington, D.C., 1960.

Bromberg, Howard, "COBOL and Compatibility," Datamation, Vol. 7, No. 2, February 1961, p. 30.

-----, "What COBOL Isn't," Datamation, Vol. 7, No. 9, September 1961, p. 27.

Burroughs Corporation, The Descriptor--A Definition of the B5000 Information Processing System, Bulletin 5000-20002-P, The Burroughs Corporation, February 1961.

Cheatham, T. E., Jr., CL-II: Notes in Implementation of the CL-II Programming, SM 61-7, The Tech/Ops Series on Programming Systems, Technical Operations, Inc., Burlington, Mass., July 1961.

-----, Preliminary Design Specifications for CL-II Programming System: Sec. 2--Syntactic and Semantic Summary of Longs L_s, L_r, L_d, Report No. TO-B-60-23, Technical Operations, Inc., Burlington, Mass., June 1960.

Cheatham, T. E., Jr., G. O. Collins, Jr., G. F. Leonard, J. W. Smith, and S. Warshall, Introduction to the CL-I Programming System, TR 59-6, Technical Operations, Inc., Burlington, Mass., January 1960.

Clippinger, R. F., "FACT - A Business Compiler Description and Comparison with COBOL and Commercial Translator," Annual Review in Automatic Programming, Pergamon Press, Jaunary 1961.

-----,"International Standards," Proceedings of the 1961 EJCC, Unpublished, December 1961.

"CODASYL O.K.'s Publication of COBOL-61--Executive Committee Resolves Not to Abdicate Maintenance," Datamation, Vol. 7, No. 7, July 1961.

Collins, G. O., Jr., and S. Warshall, _CL-II: A General Purpose Syntax Directed Compiler_, SM 61-5, The Tech/Ops Series on Programming Systems, Technical Operations, Inc., Burlington, Mass., July 1961.

Devonald, C. H., and J. A. Fotheringham, "The Atlas Computer," _Datamation_, Vol. 7, No. 5, May 1961, p. 23.

Department of Defense, _Report to Conference on Data Systems Languages--(COBOL)_, A-2270-DOD, April 1960.

Eastern Joint Computer Conference, "The Current Status of Programming Language Standardization," Proceedings of the 1961 EJCC, Panel Discussion, Unpublished, December 1961.

Electronic News, No. 303, February 19, 1962, "Data Items," p. 38.

Erdwinn, J. D., D. M. Dahm, and G. W. Logemann, _Burroughs Algebraic Compiler_, Preliminary Working Paper, Burroughs Corporation, October 1959.

Evans, Orren Y., _Advanced Analysis Method for Integrated Electronic Data Processing (IEDP)_, North American Aviation, October 1959.

Freeman, Linton C., "The Use of Simplified Programming Systems in IBM 650 Data Processing." _The Behavioral Science Journal_, February 1962.

General Electric, _GE-225 Programming Conventions_, CPB 178 (5M-9-61), General Electric Corp., Computer Department, September 1961.

-----, _GE-225 Tabsol Application Manual: Introduction to Tabsol_, CPB 147A(5M-6-61), General Electric Corp., Computer Department, June 1961.

-----, _GECOM--The General Compiler_, CPB 144(10M-4-61), General Electric Corporation, Computer Department, April 1961.

Gorn, Saul, _Some Basic Terminology Connected with Mechanical Languages and Their Processors_, Office of Computer Research and Education, University of Pennsylvania, August 1961.

-----, _The Treatment of Ambiguity and Paradox in Mechanical Languages_, Office of Computer Research and Education, University of Pennsylvania, April 1961.

Grad, Burton, "Tabular Form in Decision Logic," _Datamation_, Vol. 7, No. 7, July 1961, p. 22.

Halstead, M. H., _Machine Independent Programming_, Spartan Books, 1962.

Haverty, J. P., _The Role of Programming Languages in Command and Control: An Interim Report_, The RAND Corporation, RM-3293, September 1962.

Holt, Anatol, W., Chairman, "Proceedings, ACM Storage Allocation Symposium," Princeton, New Jersey, June 23-24, 1961, Unpublished.

Holt, A., W. Turanski, and D. L. Meginnity, Common Programming Language Task-Automatic Code Translation System (ACT), Final Report No. AD 59 URl, Institute for Cooperative Research, University of Pennsylvania, July 13, 1959.

Honeywell, FACT - A New Business Language, DSI-27A, Minneapolis-Honeywell, 1960.

Hosier, W. A., "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," IRE Transactions on Engineering Management, June 1961.

Information Processing, Proceedings of International Conference on Information Processing, UNESCO, Butterworths, London, June 1959.

Ingerman, P. Z., "Dynamic Declarations," Communications of the ACM, January 1961, pp. 59-60.

Ingerman, P., and E. Irons, THUNKS - A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations, Office of Computer Research and Education, University of Pennsylvania, November 1960.

Institute of Defense Analyses, Computers in Command and Control, Technical Report No. 61-12, November 1961.

International Business Machines, General Information Manual--IBM Commercial Translator, Form F 28-8043, International Business Machines Corp., 1960.

-----, General Information Manual--COBOL, Form 28-8052-1, International Business Machines Corp., 1960-61.

-----, "Multiprogramming--A Candid View," Unpublished Working Paper, International Business Machines Corp.

-----, Reference Manual Glossary for Information Processing, International Business Machines Corporation, 1962.

Irons, E. T., and W. Feurzeig, Comments on the Implementation of Procedures and Blocks in ALGOL-60, Office of Computer Research and Education, University of Pennsylvania, November 1960.

Jones, Graham, "Trends in Computer Hardware," Datamation, Vol. 7, No. 1, January 1961, p. 11.

Lautzenheiser, Marvin, Stage Executive Control, Technical Operations, Inc., Burlington, Mass., September 7, 1961.

Leonard, Gene F., The CL-I Programming System User's Manual, Technical Operations, Inc., Burlington, Mass., January 1961.

Lonergan, William, and Paul King, "Design of the B5000 System," Datamation, Vol. 7, No. 5, May 1961, p. 28.

Manelowitz, Howard, ANCHOR--An Algorithm for Analysis of Algebraic and Logical Expressions, SP-127, System Development Corporation, Santa Monica, Calif., November 1, 1959.

McCracken, Daniel D., A Guide to FORTRAN Programming, John Wiley and Sons, New York, 1961.

Mock, O. R., Procedures Manual for MICA, North American Aviation, February 23, 1962.

Naur, Peter, Ed., "Report on the Algorithmic Language ALGOL-60," Communications of the ACM, Vol. 3, No. 5, May 1960, p. 299.

Opler, Ascher, "Trends in Programming Concepts," Datamation, Vol. 7, No. 1, January 1961, p. 13.

Patrick, R. L., "Compiler Thoughts as of August 3, 1959," Unpublished Working Paper.

-----, "Documentation--Key to Promotion," Datamation, Vol. 7, No. 8, August 1961, p. 24.

-----, "The Gap in Programming Support," Datamation, Vol. 7, No. 5, May 1961, p. 37.

-----, "An Introduction to Automatic Programming for Business, Part 2," Data Processing, Proceedings of 1960 Convention of National Machine Accountants Association.

Porter, R. E., "The RW-400--A New Polymorphic Data System," Datamation, Vol. 6, No. 1, February 1960, p. 8.

Sammet, J. E., "More Comments on COBOL," Datamation, Vol. 7, No. 3, March 1961, p. 33.

Sattley, K., and P. Z. Ingerman, The Allocation of Storage for Arrays in ALGOL-60, Office of Computer Research and Education, University of Pennsylvania, November 1960.

SHARE, A Data Processing Compiler for the IBM 704, North American Aviation, Columbus, Ohio, July 1959.

Shaw, C. J., "JOVIAL," Datamation, Vol. 7, No. 6, June 1961, p. 29.

-----, The JOVIAL Manual, Part 1: Computer Programming Languages and JOVIAL, TM-555-1, System Development Corporation, Santa Monica, Calif., December 20, 1960.

-----, The JOVIAL Manual, Part 2: The JOVIAL Grammar and Lexicon, Rev. 1, TM-555-2, System Development Corporation, Santa Monica, Calif., June 9, 1961.

-----, The JOVIAL Manual, Part 3: The JOVIAL Primer, TM-555-3, System Development Corporation, Santa Monica, Calif., December 26, 1961.

-----, "A Programmer's Look at JOVIAL in an ALGOL Perspective," Datamation, Vol. 7, No. 10, October 1961.

Siegel, Milton, and Albert E. Smith, "Interim Report on Bureau of Ships COBOL Evaluation Program," Communications of the ACM, Vol. 5, No. 5, May 1962.

System Development Corporation, Computer Programming Standards in Command and Control, TM-688, System Development Corporation, Santa Monica, California, February 15, 1962.

Taylor, W., L. Turner, and R. Waychoff, "ALGOL-60--Syntactical Chart for the B-5000," Communications of the ACM, September 1961, p. 393.

Voorhees, E., G. L. Carter, J. Hudgins, C. Kazek, and K. Balke, "ALGAE I* - A Compiler for the IBM 704," Working Paper, May 26, 1958.

Wegstein, J. H., "ALGOL-60--A Status Report," Datamation, Vol. 7, No. 9, September 1961, p. 24.

Willey, E. L., M. Tribe, A. d'Agapeyeff, B. J. Givvens, and M. Clark, Some Commercial Autocodes: A Comparative Study, Automatic Programming Information Center, Academic Press, 1961.

Yngve, Victor H., "An Introduction to COMIT Programming," SHARE Distribution Paper, First Draft, June 12, 1961.

Young, J. W., Jr., and H. K. Kent, "Abstract Formulation of Data Processing Problems," J. of Ind. Eng., November 1958.